Scheduler-Conscious Synchronization

L.I. Kontothanassis, R.W. Wisniewski, and M.L. Scott

Technical Report 550
December 1994

DTIC
ELECTE
JAN 2 4 1995
S G D

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

19950118 081

Scheduler-Conscious Synchronization ⋅

Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226

{kthanasi,bob,scott}@cs.rochester.edu

December 1994

Unannounced	☐	
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

Abstract

Efficient synchronization is important for achieving good performance in parallel programs, especially on large-scale multiprocessors. Most synchronization algorithms have been designed to run on a dedicated machine, with one application process per processor, and can suffer serious performance degradation in the presence of multiprogramming. Problems arise when running processes block or, worse, busy-wait for action on the part of a process that the scheduler has chosen not to run.

In this paper we describe and evaluate a set of *scheduler-conscious* synchronization algorithms that perform well in the presence of multiprogramming while maintaining good performance on dedicated machines. We consider both large and small machines, with a particular focus on scalability, and examine mutual-exclusion locks, reader-writer locks, and barriers. The algorithms we study fall into two classes: those that heuristically determine appropriate behavior and those that use scheduler information to guide their behavior. We show that while in some cases either method is sufficient, in general sharing information across the kernel-user interface both eases the design of synchronization algorithms and improves their performance.

1

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE December 1994	3. REPORT TYPE AND DATES COVERED technical report

4. TITLE AND SUBTITLE

Scheduler-Conscious Synchronization

5. FUNDING NUMBERS

N00014-92-J-1801, ARPA Order 8930

6. AUTHOR(S)

L.I. Kontothanassis, R.W. Wisniewski, and M.L. Scott

MDA972-92-J-1012

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester NY 14627-0226

8. PERFORMING ORGANIZATION

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES)

Office of Naval Research   ARPA
Information Systems   3701 N. Fairfax Drive
Arlington VA 22217   Arlington VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

TR 550

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS

synchronization; scheduler-conscious synchronization; preemption-safe synchronization

15. NUMBER OF PAGES

47 pages

16. PRICE CODE

free to sponsors; else $2.00

17. SECURITY CLASSIFICATION OF REPORT unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT unclassified	20. LIMITATION OF ABSTRACT UL

1 Introduction

One of the most basic questions for any synchronization mechanism is whether a process that is unable to continue should *spin*—repeatedly testing the desired condition—or *block*—yielding the processor to another, runnable process. Early work addressed this question primarily in the context of uniprocessors. For such machines, spinning ("busy-waiting") makes sense for synchronization among the memory operations of the processor and its devices, but scheduler-based (blocking) synchronization must be used when the synchronizing processes are being multiplexed on top of a single processor.

With the advent of shared-memory multiprocessors, busy-wait synchronization has also come to be widely used in user-level applications. Spinning makes sense when the expected wait time of a synchronization operation is less than twice the context switch time, or when the spinning processor has nothing else useful to do. Researchers have developed a wealth of busy-wait mechanisms, including mutual exclusion locks, reader-writer locks (which allow concurrent access among readers, but guarantee exclusive access by writers), and *barriers* (which guarantee that no process continues past a given point in a computation until all other processes have reached that point). Of particular interest in recent years have been *scalable* synchronization algorithms, which employ backoff or distributed data structures to minimize contention [1, 9, 11, 19, 20, 25, 26, 30, 31, 32, 38, 44, 45]. (The purpose of backoff is to reduce the frequency with which spinning processes access a common synchronization variable. The purpose of distributed data structures is to allow each process to spin on a separate, locally-accessible variable.)

Unfortunately, busy-waiting in user-level code tends to work well only if each process runs on a separate physical processor. If the total number of processes in the system exceeds the number of processors, them some processors will have to be multiprogrammed. The processes on a given processor may be from different applications or, if the scheduler partitions the machine, from a single application. In either case, conflicts between scheduling and synchronization can seriously degrade performance when:

- a process is preempted while holding a lock,

- a process is preempted while waiting for a lock and then is handed the lock while still preempted, or

- a process spins when some preempted process could be making better use of the processor.

In our experiments, we have found that algorithms that provide excellent performance in the absence of multiprogramming may perform orders of magnitude worse when multiprogramming is introduced. These results suggest the need for *scheduler-conscious* synchronization—techniques that use knowledge of the scheduler's actions when making synchronization decisions, and that provide information to the scheduler that it can use when making its decisions.

Several research groups have addressed one or more aspects of scheduler-conscious synchronization. Some have shown how to avoid preempting a process that holds a `test_and_set` lock, or to recover from this preemption if it occurs. Others have developed heuristics

2

that allow a process to guess whether it would be better to relinquish the processor, rather than spin, while waiting for a lock. Several important aspects, however, have not yet been addressed. In this paper we provide a relatively comprehensive treatment of scheduler-conscious synchronization, covering mutual exclusion locks, reader-writer locks, and barriers, for both large and small machines. Our contributions include:

- scheduler-conscious ticket and list-based queue locks, which provide FIFO servicing of requests and scale well to large machines;

- a fair, scalable, scheduler-conscious reader-writer lock (the non-scalable version is trivial);

- a scheduler-conscious barrier for small machines (in which a centralized data structure does not suffer from undue contention, and in which processes can migrate between processors); and

- a scheduler-conscious barrier for large machines that are partitioned among applications, and on which processes migrate only when repartitioning.

The key to most of our techniques is to widen the kernel/user interface. We propose three simple interface extensions that inform an application about the status of its processes and the processors on which they run, and that allow it, safely and within limits, to limit the points at which preemption may occur. Using this kernel interface, our algorithms are able to achieve performance that degrades gracefully and linearly with increases in the level of multiprogramming, while scaling without contention to very large numbers of processors. We also consider heuristic techniques that attempt to adapt to the scheduling policy without widening the kernel interface. These techniques achieve acceptable performance in certain cases, but in general we find that use of the wider interface results in cleaner code and better performance.

We assume the availability of special instructions that allow a process to read, modify, and write a shared variable as a single atomic operation.[1] Examples include test_and_set, fetch_and_increment, fetch_and_store (swap), compare_and_store, and the pair load_-linked and store_conditional [15], now available in the DEC Alpha and MIPS II architectures. The atomic instructions used in our algorithms can all be emulated efficiently by load-linked and store-conditional.

Our emphasis on large machines stems not only from the fact that scalable algorithms are newer and hence less studied, but also from the fact that scheduler-conscious synchronization is inherently *harder* for scalable algorithms based on distributed data structures. Scalable algorithms have difficulties because their deterministic ordering of processes can conflict with the actions of the scheduler in a multiprogrammed system. For example: a mutual exclusion lock may keep waiting processes in a FIFO queue, either for the sake of fairness

---

[1]Some multiprocessors, especially the larger ones, provide more sophisticated hardware support for synchronization. Examples include the queued locks of the Stanford Dash machine [21], the QOLB (queue-on-lock-bit) operation of the IEEE Scalable Coherent Interface [14], and the near-constant-time barriers of the Thinking Machines CM-5 and the Cray Research T3D. It is not yet clear whether the advantages of such special operations over simpler read-modify-write instructions are worth the implementation cost.

or to minimize contention. The algorithm's performance is then vulnerable not only to preemption of the process in the critical section, but also to preemption of processes near the head of the waiting list—the algorithm may give the lock to a process that is not running [48]. Similarly, a barrier algorithm may keep processes in a tree, in order to replace $O(n)$ serialized operations on a counter with $O(\log n)$ operations on the longest path in the tree. But then processes must execute their portions of the barrier algorithm in the order imposed by the tree. If the processes on a given processor are scheduled in a different order, and if they simply yield the processor when unable to proceed (as opposed to waiting on a kernel-provided synchronization queue), then the scheduler may need to cycle through most of the ready list *several times* in order to achieve a barrier [7].

The rest of the paper is organized as follows. Section 2 discusses related work. It explains in more detail why synchronization algorithms suffer under multiprogramming, why scalable synchronization algorithms are particularly susceptible to multiprogramming effects, and why previous research does not fully remedy the problem. Section 3 describes our kernel interface and compares it to alternative approaches, such as process-to-process hand-shaking and experience-based heuristics, that use a more conventional interface. Section 4 describes our scheduler-conscious algorithms, in prose and pseudo-code. Section 5 describes our experimental environment and presents performance results. Conclusions appear in section 6.

2   Background

In this section we describe sources of performance loss due to the adverse interaction of scheduling and synchronization. We discuss related work regarding multiprogramming environments, synchronization algorithms, and their interaction, while placing our work in context.

2.1   Small-Scale Locks

It is widely recognized that lock-based algorithms (i.e. mutual exclusion and reader-writer locks) can suffer performance losses when a process is preempted while in a critical section. Remaining processes cannot access the shared data structure or protected resource until the preempted process releases the lock it is holding. (In a reader-writer lock, a preempted reader will only prevent writers from accessing the shared data structure. In a fair implementation this has the side effect that readers following the writers are also denied access.)

Ousterhout [35] introduced *spin-then-block* locks that attempt to minimize the impact of preemption (or other sources of delay) in critical sections by having a waiting process spin for a small amount of time and then, if unsuccessful, block. Karlin et al. [16] present and evaluate a richer set of spin-then-block alternatives, including *competitive* techniques that adjust the spin time based on past experience.[2] Their goal is to adapt to variability in

---

[2]A competitive algorithms is one whose worst-case performance is provably within a constant factor of optimal worst-case performance.

the length of critical sections, rather than to cope with preemption. Competitive spinning works best when the behavior of a lock does not change rapidly with time, so that past behavior is an appropriate indicator of future behavior.

Zahorjan et al. [46, 48] present a formal model of spin-wait times. For lock-based applications in which all processes on a given processor belong to the same application, they show that performance problems can be avoided if the operating system simply partitions processes among processors and allows the application to make intra-processor scheduling decisions (never preempting a process with a lock).

Several groups have proposed extensions to the kernel/user interface that allow a system to avoid adverse scheduler/lock interactions while still doing scheduling in the kernel. The Scheduler Activation proposal of Anderson et al. [2] allows a parallel application to recover from untimely preemption. When a processor is taken away from an application, another processor in the same application is given a software interrupt, informing it of the preemption. The second processor can then perform a context switch to the preempted process if desired, e.g. to push it through its critical section. In a similar vein, Black's work on Mach [5] allows a process to suggest to the scheduler that it be de-scheduled in favor of some specific other process, eg. the holder of a desired lock. Both of these proposals assume that process migration is relatively cheap.

Rather than recover from untimely preemption, the Symunix system of Edler et al. [8] and the Psyche system of Marsh et al. [29] provide mechanisms to avoid or prevent it. The Symunix scheduler allows a process to request that it not be preempted during a critical section, and will honor that request, within reason. The Psyche scheduler provides a "two-minute warning" that allows a process to estimate whether it has enough time remaining in its quantum to complete a critical section. If time is insufficient, the process can yield its processor voluntarily, rather than start something that it may not be able to finish.

2.2 Alternative Approaches to Atomic Update

All of the work in the previous section aims to make spin locks safe from untimely preemption. An alternative approach is to avoid the use of locks. There are at least two ways of doing so.

Herlihy [12, 13] has led the development of *lock-free* and *wait-free* data structures. Rather than rely on locks, these data structures use special algorithms based on such *universal* atomic primitives as compare_and_store and load_linked/store_conditional. The algorithms are designed in such a way as to guarantee both atomicity and forward progress, despite arbitrary delays on the part of individual processes.[3] The key idea in most of these algorithms is to modify a copy of (a portion of) the data structure, and then swap it for the original in one atomic step (assuming the original has not been modified since the copy was created). Tolerance of arbitrary delays means that lock-free and wait-free data structures

---

[3]Wait-free data structures are starvation-free; lock-free data structures are not. More formally, a data structure is wait-free if every process that attempts to perform an atomic update is guaranteed to succeed in a bounded number of steps. A data structure is lock-free if *some* competing process is guaranteed to complete an operation in a bounded number of steps. An atomic primitive is universal if a constant number of instances of it can be used to emulate any other single-word atomic primitive.

are immune to the performance effects of inopportune preemption. It also means that they can tolerate some page faults and even certain kinds of hardware failure, something none of the techniques in the previous section can do. Unfortunately, the current state of the art in general-purpose lock-free and wait-free synchronization techniques incurs substantial performance overhead, even where there is no competition for access to the data structure.

A second way to avoid the use of locks is to create a manager process that is responsible for all operations on the "shared" data structure, and to require other processes to send messages to the manager. This sort of organization is common in distributed systems. It can be cast as a natural interpretation of monitors (as, for example, in Brinch Hansen's *Distributed Processes* notation [6]), or as *function shipping* [24, 40] to a common destination. In recent years, several machines have been developed that provide hardware support for very fast invocation of functions on remote processors [18, 33]. Even on more conventional hardware, programming techniques such as *active messages* [42] can make remote execution very fast. Because computation is centralized and requests are processed serially, active messages provide implicit synchronization. On the other hand, they do not permit concurrency (as do, say, reader-writer locks), and can only be used when the manager is not a bottleneck. They also require that data be accessed through a narrow interface (the data are not directly visible), and are inefficient on some machines.

2.3 Scalable Locks

When two processes spin on the same location, coherence operations or remote memory references (depending on machine type) can create substantial amounts of contention for memory and for the processor-memory interconnect. The key to good performance is to minimize active sharing. One option is to use backoff techniques [1, 30] in which a processor that attempts unsuccessfully to acquire a lock waits for a period of time before trying again. The amount of time depends on the estimated level of contention. Bounded exponential backoff works well for test_and_set locks. Backoff proportional to the number of predecessors works well for ticket locks.

A second option for scalable locks is to use distributed data structures to ensure that no two processes spin on the same location. The queue-based spin locks of Anderson [1] and of Graunke and Thakkar [9] minimize active sharing on coherently-cached machines by arranging for every waiting processor to spin on a different element of an array. Each element of the array lies in a separate, dynamically-chosen cache line, which migrates to the spinning processor. The queue-based spin lock of Mellor-Crummey and Scott [30] represents its queue with a distributed linked list instead of an array. Each waiting processor uses a fetch_and_store operation to obtain the address of the list element (if any) associated with the previous holder of the lock. It then modifies that list element to contain a pointer to its *own* element, on which it then spins. Because it spins on a location of its own choosing, a process can arrange for that location to lie in local memory even on machines without coherent caches. Magnussen et al. [26] have shown how to modify linked-list queue-based locks to minimize interprocessor communication on a coherently-cached machine. Markatos [27] uses a doubly-linked list in a queue-based lock to replace FIFO order with highest-priority-first, for real-time systems. Others have shown how to build queue-based

scalable reader-writer locks[19, 31]. Yang and Anderson [44] have shown how to use local-only spinning to improve the performance of locks that use no atomic memory operations other than loads and stores.

Scalable synchronization algorithms reduce contention for memory and interconnect resources but usually have higher overhead than their centralized counterparts in the absence of contention. Lim and Agarwal [23] suggest synchronization algorithms that monitor the degree of contention and switch between centralized and distributed implementations based on the observed degree of competition for synchronization variables.

Moving from a centralized to a queue-based lock adds one more dimension to the scheduler/synchronization problem. In order to have every process spin on a separate variable, queue-based locks require that processes acquire the lock in a deterministic (generally FIFO) order. If a process is preempted while awaiting its turn to access a shared data structure, processes later in the order cannot proceed even if the lock is released by the original owner—the lock will be passed to the preempted process instead. This problem was noted by Zahorjan et al. [48], but no solution was suggested. In section 4 we present two mutual exclusion locks and a reader-writer lock that solve the problem by bypassing preempted processes in the queue and having them retry for the lock when they resume execution.[4]

2.4 Barriers

Barrier synchronization algorithms force processes to wait at a specified point in the computation until all their peers have arrived at that same point. From a scheduling point of view, the principal difference between locks and barriers is that while the time between lock acquisition and release is generally bounded and short (one critical section's worth of computation), the time between consecutive barriers can be arbitrarily long. This means, for example, that while it may be acceptable to disable preemption in a process that holds a lock, it is not acceptable to do so in a process that must continue to execute in order to reach the next barrier.

Performance loss in barriers occurs when processes spin uselessly, waiting for preempted peers. When a process on a multiprogrammed processor spins at a barrier that has not yet been reached by some other process on the same processor, it may waste as much as a quantum, reducing system throughput and likely increasing the computation's critical path length.

Inspired by Karlin et al., we have developed techniques to make the spin versus block decision at a centralized (small-scale) barrier [17]. We present these techniques in the beginning of section 4.3. They are most useful in an environment in which the OS-level scheduler partitions processors among applications (so that multiprogramming is only among the processes of a single application), but they can be used in a more general time-shared environment as well. Heuristics based on past behavior suffice to obtain competitive performance, but a small extension to the kernel/user interface admits on-line optimal decisions.

---

[4]The mutual exclusion locks originally appeared in a conference publication [43].

Centralized barriers generally employ a counter that is updated by each process. Access to the counter poses two obstacles to scalable performance on large machines. First, as with locks, simultaneous attempts to update the counter can lead to unacceptable levels of contention. Second, even in the absence of contention, serial access to a counter implies an asymptotic running time of $O(p)$ for the barrier algorithm, which becomes unacceptable as the number of processors p grows large. Several researchers have shown how to solve these problems by building *scalable* barriers, with log-depth tree- or FFT-like patterns of point-to-point notifications among processes [3, 11, 20, 25, 30, 32, 38, 45].

Unfortunately, the deterministic notification patterns of scalable barriers may require that processes run in a different order from the one chosen by the scheduler. The problem is related to, but more severe than, the preemption-while-waiting problem in FIFO locks. With a lock the scheduler may need to cycle through the entire ready list before reaching the process that is able to make progress. With a scalable busy-wait barrier, Markatos et al. have shown [28] that the scheduler may need to cycle through the entire ready list a logarithmic number of times (spinning for as long as a quantum between context switches) in order to achieve the barrier. To avoid this problem, they suggest (without an implementation) that blocking synchronization be used among the processes on a given processor, with a scalable busy-wait barrier among processors. (Such *combination* barriers were originally suggested by Axelrod [4] to minimize resource needs in barriers constructed from OS-provided locks.) The challenge for a combination barrier is to communicate partition information from the scheduler to the application, and to adapt to partitioning changes at run time. We present such a barrier in the latter part of section 4.3, enhanced with our optimal spin versus block decision-making technique within each processor.

2.5   Multiprogramming on Multiprocessors

Multiprogramming on uniprocessors is desirable because it minimizes response time and makes effective use of system resources. Simple timesharing allows multiple users to share a processor transparently, with only a small performance penalty. Multiprocessors and parallel programs complicate maters significantly, by introducing synchronization and by requiring that performance metrics consider issues such as speedup. Simple timesharing may lead to very bad performance if processes that need to synchronize with one another do not run simultaneously, or if an application runs for a small amount of time on a large number of processors (at low efficiency) rather than for a large amount of time on a small number of processors (at high efficiency).

To address the synchronization issue, Ousterhout et al. [34] developed the notion of *co-scheduling* (also known as *gang scheduling*), in which all of the processes of an application run at the same time. Unfortunately, bin-packing problems make it difficult to use all processors effectively in a co-scheduled system, and context switches among applications detract from computation time and lead to sub-optimal use of caches.

Using simulation, modeling, and experimentation, Crovella et al. [7], Leutenegger and Vernon [22], Tucker and Gupta [10], and Zahorjan and McCann [47] have shown that dynamically partitioning the processors of a machine among applications is preferable to either timesharing or co-scheduling. Tucker and Gupta also identify the existence of more

8

	Centralized	Scalable
Locks	1, 3	1, 2, 3
Barriers	3	3

Table 1: Types of performance loss encountered in different synchronization algorithms

processes than processors as the main reason for remaining performance degradation and propose a scheme in which applications adapt to changes in the size of hardware partitions by changing the number of processes they use [41]. In a similar vein, Zahorjan et al. [48] suggest that barrier-based programs use guided self-scheduling [36], with exactly one server thread per processor. It is not clear, however, that all applications will be able to adjust their number of processes dynamically, or that programmers will wish to write in a style that allows them to do so. Our combination barrier is designed to work well even for applications with a fixed number of processes on a dynamically changing number of processors.

2.6 Summary

In section 1 we listed three scenarios under which programs can suffer performance loss due to adverse interactions between scheduling and synchronization:

1. A process is preempted while holding a lock. The critical path of the computation may increase if other processes (on the same or different processors) need to acquire the lock.

2. A process is preempted while waiting for a lock and then is handed the lock while still preempted. As in (1), critical path length may increase.

3. A process spins when some preempted process could make better use of the processor. This is the flip side of lock scenarios (1) and (2). It also arises in barriers. Critical path length may increase, and overall system throughput suffer.

Table 1 shows which of these scenarios are possible for each type of synchronization (centralized/scalable, locks/barriers). Previous work has produced satisfactory solutions for the case of centralized locks but only partial solutions in the other three cases. In the following sections we show how to build

- scalable scheduler-conscious queue-based and ticket locks;

- a scheduler-conscious reader-writer lock;

- scheduler-conscious centralized barrers that make optimal spin versus block decisions on small time-shared machines; and

- scheduler-conscious scalable barriers for large, partitioned machines.

9

3 Solution Structure

In order to avoid adverse interactions, either the scheduler or the synchronization algorithm could adapt to the behavior of the other. We assume that user-level code can influence the behavior of the scheduler enough to avoid preemption in (short) critical sections, but that otherwise the scheduler is in control, and it is the synchronization algorithm that must adapt. Scheduler-conscious synchronization algorithms therefore require a mechanism to obtain information about the status of other processes and about the processors available to the application. This information may be (1) guessed via past experience using heuristics, (2) deduced through interaction with other processes, e.g. via "handshaking", or (3) provided by the kernel itself. In order, these options provide information of increasing accuracy. They also incur increasing costs in performance and/or complexity: experience-based heuristics require only the collection of process-local statistics; process interactions take time, and lead to more complicated code; kernel-provided information requires changes to the kernel/user interface.

Experience-based heuristics can be successful to the extent that the present and future resemble the past. It forms the basis of the competitive lock algorithms of Karlin et al. [16], and of the competitive barriers we describe at the beginning of section 4.3. In the case of locks, the goal is to block if the wait time will be longer than twice the context switch time, and to spin if it will be shorter. For barriers, the goal is to block if there is another process (not currently running) that could use the current processor to make progress toward the barrier, and to spin otherwise. In both cases, the algorithm is able to determine (by reading the clock) whether blocking or spinning would have been a better policy at the most recent synchronization operation. If it finds it made the wrong decision, it biases its decision in favor of the other alternative the next time around. While this sort of adaptation has been shown to work better than any static alternative, it induces overhead to maintain the statistics that allow the decision to be made, and still makes the wrong decision some of the time. In the case of delay variance due to preemption, it is not clear that recent behavior is a particularly good predictor.

Interactions with peer processes can provide better information about the peers' status, provided that they respond promptly to enquiries (when running). In section 4.1, for example, we use a "handshaking" technique in some of our mutual exclusion algorithms. To hand a peer a lock, a process sets a flag on which the peer is expected to be spinning, and then waits for the peer to set an acknowledgment flag. If the acknowledgment does not appear within a certain amount of time, the signaling process assumes that the peer is currently preempted. There is an inherent inefficiency with this approach however: if the signaling process doesn't wait long enough, it will too often skip over a running peer by mistake. Any time it waits, however, is lost to computation. All the signaling process really needs to know is whether its peer is running or preempted, information readily available to the scheduler.

We have found that using heuristics or handshaking to determine the state of a process or group of processors can be expensive both in implementation and execution cost. We therefore propose extensions to the kernel interface to make appropriate information available to the user. Algorithms based on kernel-provided information are simpler and easier to

10

design. They also tend to perform better, in part because the information from the kernel is more accurate than user-level estimates, and in part because the kernel can collect the information more efficiently than user-level code can guess it.

Our kernel extensions are enumerated below. They build upon ideas proposed by the Symunix project at NYU [8]. Similar extensions could be based on the kernel interfaces of Psyche [29] or Scheduler Activations [2].

- KE-1: For each process the kernel and user cooperate to maintain a variable that represents the process's state and that can be manipulated under certain rules by either. The variable has four possible values: `preemptable`, `preempted`, `self_unpreemptable`, and `other_unpreemptable`. Preemptable indicates that the process is running, but that the kernel is free to preempt it. Preempted indicates that the kernel has preempted the process. Self_unpreemptable and other_unpreemptable indicate that the process is running and should not be preempted. The kernel honors this request whenever possible (see KE-2 below), deducting any time it adds to the end of the current quantum from the beginning of the next.

  The `non-preemptable` states indicate that the process is executing in a critical section. The need for two distinct states arises from the need to accommodate certain race conditions in queue-based mutual exclusion algorithms, in which a process wishes to hand the lock to one of its peers and simultaneously make that peer unpreemptable. Most changes to the state variable make sense only for a particular previous value. For example, it makes no sense for user-level code to change a state variable from `preempted` to anything else. Overall system correctness does not depend on correct use of flags by applications, but the performance of a particular application may suffer if it uses the flags incorrectly. To make sure that changes happen only from appropriate previous values, our algorithms generally modify state variables using an atomic `compare_and_store` instruction.[5]

- KE-2: To ensure fairness for applications, the kernel maintains an additional per-process Boolean flag. This flag can be modified only by the kernel but is readable in user mode. The kernel sets a process's flag to indicate that it wanted to preempt the process but has honored a request (indicated via the KE-1 variable) not to do so. To maintain ultimate control of the processor, the kernel honors the request only when the KE-2 flag is not yet set; if the flag is already set the kernel proceeds with the preemption. Upon exiting a critical section (and setting the KE-1 variable back to `preemptable`), a process should inspect the flag and yield the processor if the flag is set. Yielding implicitly clears the flag. So long as critical sections are shorter than the interval between the kernel's attempts at preemption, voluntarily yielding the processor at the end of critical section in which the KE-2 flag has been set ensures that preemption will not occur during a subsequent critical section (barring page faults or other unusual sources of delay).

---

[5]Compare_and_store (`location`, `expected_val`, `new_val`) compares the value `expected_val` to the contents of `location`. If they are identical it stores `new_val` in `location` and returns `true`. Otherwise it returns `false`.

11

- KE-3: The kernel also maintains a data structure, visible in user mode, that contains information about the hardware partition on which the application is running. Specifically, the information maintained includes the number of processors available in the partition, the id of the current processor, the number and ids of processes scheduled on each processor, and the *generation count* of the partition. The *generation count* indicates the number of times that the partition has changed in size since the application started running.

As noted above, extensions KE-1 and KE-2 are based in part on ideas developed for the Symunix kernel [8]. We have introduced additional states, and have made the state variable writable and readable by both user-level and kernel-level code [43]. Extension (KE-3) is a generalization of the interface described in our work on small-scale scheduler-conscious barriers [17] and resembles the "magic page" of information provided by the Psyche kernel [37]. None of the extensions requires the kernel to maintain information that it does not already have available in its internal data structures. Furthermore, the kernel requires no knowledge of the particular synchronization algorithm(s) being used by the application, and does not need to access any user level code or data structures. We have run our experiments in user space, but a kernel-level implementation of our ideas would not be hard to build.

Like most scalable synchronization algorithms, ours achieve their scalability by arranging for processors to spin only on local locations, on which no other processor spins. We also endeavor to ensure that those locations will be local not only on cache coherent machines (on which they migrate to the spinning processor), but also on machines that lack hardware cache coherence. On these latter, NUMA machines, variables on which processes spin must be allocated statically in the local memory of the spinning processor; spins are terminated by a single uncached remote write by another processor. In several cases we achieve this static allocation by means of indirection. Given the following code for a cache-coherent machine:

```
shared f : Boolean := false
...
process A:
    repeat until f
...
process B:
    f := true
```

we can employ the following version on a non-cache-coherent machine:

```
const available := // some non-nil bit pattern that is not a valid pointer
shared f : ^Boolean := nil
...
process A:
    private my_flag : Boolean := false
    private temp : ^Boolean := fetch_and_store (f, &my_flag)
    if temp != available
```

```
        repeat until my_flag
   ...
   process B:
       private temp : ^Boolean := fetch_and_store (f, available)
       if temp != nil
           temp^ := true
```

In rare cases, the overhead of indirection may not be warranted. There is one point in our reader-writer lock, for example, at which an unlikely race condition may force a process to spin until a pointer on another processor becomes non-nil. We could employ a method similar to that above to eliminate the remote spin, but the cumulative overhead in the common case would exceed the cumulative savings in the uncommon case.

4   Algorithms

In this section we present scheduler-conscious synchronization algorithms that make use of heuristics, handshaking, and the extended kernel interface described in section 3. We consider mutual exclusion, reader-writer locks, and barriers in turn.

The pseudocode in figure 1 defines the interface between the kernel and the application. The state field of a context_block is written by application processes to indicate when they do not want to be preempted. The remaining fields of both the context_block and partition_block records are writable only by the kernel scheduler; they provide the application with information about system state, to facilitate the design of efficient algorithms. Our mutual exclusion and reader-writer locks use only the context_block records; the barriers use both.

All of our algorithms work well in a dynamic hardware-partitioned environment—an environment widely believed to provide the best combination of throughput and fast turnaround for large-scale multiprocessors [7, 22, 41, 47]. Except for the barriers, which require partition information, all of the algorithms will also work well under ordinary time sharing. For a co-scheduled environment the additional complexity of scheduler-conscious algorithms is not necessary, but does not introduce any serious overhead.

```
type context_block = record
    state : (preempted, preemptable, unpreemptable_self, unpreemptable_other)
    warning : Boolean
    ...

type partition_block = record
    num_processors, generation : integer
    processes_on_processor : array [MAX_PROCESSORS] of integer
    processor_ids : array [MAX_PROCESSES] of integer
    ...
```

Figure 1: Pseudocode declarations for the kernel-application interface.

13

4.1 Mutual Exclusion

Scalability in mutual exclusion algorithms is best achieved by arranging for processes to spin on local locations, thereby eliminating interconnect and memory contention. Many researchers have now developed algorithms of this type [1, 9, 26, 30, 44]. The scalability of centralized algorithms may also be improved by introducing appropriate forms of back-off [1, 30]. The scheduler-conscious test_and_set locks of Psyche, Symunix, or Scheduler Activations can be modified trivially to incorporate backoff, though the work of Anderson and of Mellor-Crummey and Scott suggests that the result will still produce more contention than a queue-based lock. In this section we present two scheduler-conscious variants of the queue-based lock of Mellor-Crummey and Scott. We also present a scheduler-conscious variant of the ticket lock. This latter lock, while less trivial than a scheduler-conscious test_and_set lock, is substantially simpler than a queue-based lock, and is likely to provide acceptable performance for many environments. Unlike a test_and_set lock, the ticket lock also provides fair FIFO ordering among currently-running processes. Because they awaken processes in a deterministic order, both the queue-based and ticket locks must be modified to address not only preemption within a critical section, but also preemption while waiting in line.

Our first variant of the queue-based lock uses the Symunix kernel interface: kernel extension KE-2 and the preemptable and self_unpreemptable values (only) of KE-1. It uses handshaking to determine the status of other processes. When releasing a lock, a process notifies its successor in the queue that it (the successor) is now the holder of the lock. The successor must then acknowledge receipt of the lock by setting another flag. If this acknowledgement is not received within a fixed amount of time, the releasing process assumes that its successor is preempted, rescinds its notification, and proceeds to the following process (throughout this period the releasing process is unpreemptable).

Atomic fetch_and_store instructions are used to access the notification flag in order to avoid a timing window that might otherwise occur if the successor were to see its notification flag just before the releaser attempts to rescind it. Without the atomic instruction it would be possible for the releaser to think that the successor has failed and proceed to give the lock to another processor, and for the successor to think that it has succeeded and proceed to the critical section, thus violating mutual exclusion. Pseudocode for the handshaking algorithm appears in figure 2.

The handshaking version of the queue-based lock solves the preemption problem but unfortunately adds significant overhead to the common case. Processes need to interact several times when a lock is released. To address this limitation, we have designed a scheduler-conscious algorithm that uses the full version of kernel extension KE-1 and does not require handshaking. In this *Smart Queue* algorithm the releasing process examines its successor's state variable, which is kept up-to-date by the kernel. If the successor is preempted, the releaser proceeds to other candidates later in the queue. If the successor is running, the releaser uses an atomic compare_and_store instruction to change the successor's state to other_unpreemptable. If the change is successful the lock is passed to the successor. The need for compare_and_store stems from a potential race between the releaser and the kernel: after determining that the successor is not preempted, we must make it unpreemptable

14

```
type multi_flag = (not_yet, can_go, got_it, lost_it, ack, nack)
type qnode = record
     next, prev : ^qnode
     next_done : Boolean
     status : multi_flag
type lock = ^qnode
private cb : ^context_block

procedure acquire_lock (L : ^lock, I : ^qnode)
     loop
          I->next := nil
          cb->state := unpreemptable_self
          I->prev := fetch_and_store (L, I)
          if I->prev = nil   return
          I->status := not_yet
          I->prev->next := I
          repeat
               cb->state := preemptable
               if cb->warning  yield            // kernel wanted to preempt me
               cb->state := unpreemptable_self
          until I->status != not_yet            // spin
          val : multi_flag := fetch_and_store (I->status, got_it)
          if val = can_go
               I->prev->next_done := true        // tell prev I'm done with its qnode
               repeat until I->status = ack      // let prev finish using my qnode
               return
          while val != nack val := I->status  // wait until qnode no longer needed

procedure release_lock (L : ^lock, I: ^qnode)
     if I->next = nil                            // no known successor
          if compare_and_store (L, I, nil)  goto rtn
          repeat while I->next = nil             // spin
     I->next_done := false
     loop
          I->next->status := can_go
          for i in 1..TIMEOUT                     // spin
               if I->next_done
                    I->next->status := ack;  goto rtn
          if fetch_and_store (I->next->status, lost_it) = got_it
               // oh! successor was awake after all
               repeat until I->next_done
               I->next->status := ack;  goto rtn
          succ : ^qnode := I->next->next         // successor was asleep
          if succ = nil
               if compare_and_store (L, I->next, nil)
                    I->next->status := nack;  goto rtn
               repeat while (succ := I->next->next) = nil  // spin; non-local
          I->next->status := nack
          I->next := succ;  succ->prev := I
rtn:
     cb->state := preemptable
     if cb->warning  yield                       // kernel wanted to preempt me
```

Figure 2: Queued Handshake lock using the Symunix kernel interface.

15

```
type qnode = record
    self : ^context_block
    next : ^qnode
    next_done : Boolean
    status : (waiting, success, failure)
type lock = ^qnode
private cb : ^context_block;

procedure acquire_lock (L : ^lock, I : ^qnode)
    repeat
        I->next := nil
        I->self := cb
        cb->state := unpreemptable_self
        pred : ^qnode := fetch_and_store (L, I)
        if pred = nil
            return
        I->status := waiting
        pred->next := I
        (void) compare_and_store (&cb->state,
                                  unpreemptable_self, preemptable)
        repeat while I->status = waiting        // spin
    until I->status = success

procedure release_lock (L : ^lock, I : ^qnode)
    shadow : ^qnode := I
    candidate : ^qnode := I->next
    loop
        if candidate = nil
            if compare_and_store (L, shadow, nil)
                exit loop                       // no one waiting for lock
            repeat while shadow->next = nil     // spin; probably non-local
            candidate := shadow->next
        // order of following checks is important
        if compare_and_store (&candidate->self->state,
                              unpreemptable_self, unpreemptable_other)
            or compare_and_store (&candidate->self->state,
                                  preemptable, unpreemptable_other)
            candidate->status := success
            exit loop
        // else candidate seems to be preempted
        shadow := candidate                     // move down queue
        candidate := shadow->next
        shadow->status := failure
    cb->state := preemptable
    if cb->warning
        yield
```

Figure 3: Smart Queue lock using kernel extensions KE-1 and KE-2.

16

without giving the kernel an opportunity to preempt it. Pseudocode for the Smart Queue lock appears in figure 3.

One of the problems with queue-based locks is high overhead in the absence of contention. On small-scale machines and for low-contention locks a test_and_set with exponential backoff or ticket lock with proportional backoff may be preferable [30]. (A hybrid lock that switches between test_and_set and a queue-based lock, depending on observed contention, is another possibility [23].) With appropriate backoff, test_and_set and ticket locks scale equally well. They use different atomic instructions, making them usable on different machines. The ticket lock also guarantees FIFO service (relaxed in our scheduler-conscious version to FIFO among those processes currently running), while the test_and_set lock admits the possibility of starvation.

The basic idea of the ticket lock is reminiscent of the "please take a number" and "now serving" signs found at customer service counters. When a process wishes to acquire the lock it performs an atomic fetch_and_increment on a "next available number" variable. It then spins until a "now serving" variable matches the value returned by the atomic instruction. To avoid contention on large-scale machines, a process should wait between reads of the "now serving" variable for a period of time proportional to the difference between the last read value and the value returned by the fetch_and_increment of the "next available number" variable. To release the lock, a process increments the "now serving" variable.

Our scheduler-conscious, handshaking version of the ticket lock (figure 4) uses one additional "acknowledgment" variable, which contains the number of the last granted but unacknowledged ticket. A releasing process sets the additional variable and the "now-serving" variable and waits for the former to be reset. If the acknowledgment does not occur within a timeout window, the releaser withdraws its grant of the lock, and re-increments the "now serving" variable in an attempt to find another acquirer. Changes to the acknowledgment variable are made with compare_and_store to avoid an update race between a skipped-over acquirer and its successor. Our ticket lock assumes that the "now serving" variable does not have the opportunity to wrap all the way around and reach a value it previously had, while a process remains preempted. For 32-bit integers, a 1 GHz processor, and an empty critical section, a process would have to be preempted for more than three minutes before correctness would be lost. Going to 64-bit integers would extend this time to over 16,000 years.

There is no obvious way to develop a scheduler-conscious version of the ticket lock without either handshaking or exporting lock code into the kernel. The problem is that the lock does not keep track of the identities of waiting processes. The releaser of a lock is therefore unable to use KE-1 to determine the status of its successor: it does not know who the successor is.

A caveat with all three of our scheduler-conscious locks is that they give up the FIFO ordering of the scheduler-oblivious version. It is thus possible (though highly unlikely) that a series of adverse scheduling decisions could cause a process to starve. We have considered algorithms that leave preempted processes in an explicit queue so that they only lose their turn while they are preempted. Markatos adopted a similar approach in his real-time queued lock [27], where the emphasis was on passing access to the highest-priority waiting process. For simple unprioritized mutual exclusion, leaving preempted processes in the queue mainly

```
type t_lock = record
    next_ticket, now_serving, rel_flag : unsigned integer
private cb : ^context_block

procedure acquire_lock (L : ^t_lock)
restart:
    cb->state := unpreemptable_self
    my_ticket : integer := fetch_and_increment (&L->next_ticket)
        // overflow is benign
    repeat
        cb->state := preemptable
        if cb->warning
            yield
        cb->state := unpreemptable_self
        if (my_ticket - L->now_serving) > MAX_PROCESSES
            // I've been passed up (overflow is benign)
            goto restart
        for i in 1..((my_ticket - L->now_serving) * SPIN_FACTOR)
            // spin
    until my_ticket = L->now_serving
    if !compare_and_store (&L->rel_flag, my_ticket, my_ticket-MAX_PROCESSES)
        goto restart

procedure release_lock (L : ^t_lock)
retry:
    new_ticket : integer := L->rel_flag := L->now_serving + 1
    L->now_serving := new_ticket
    if L->next_ticket = L->now_serving      // nobody waiting
        goto rtn
    for i : integer in 1..TIMEOUT
        if L->rel_flag = new_ticket - MAX_PROCESSES
            goto rtn
    // we timed out
    if compare_and_store (&L->rel_flag, new_ticket, new_ticket-MAX_PROCESSES)
    // ticket successfully rescinded
        goto retry
rtn:
    cb->state := preemptable
    if cb->warning
        yield
```

Figure 4: Scheduler-conscious ticket lock.

makes the common case more expensive: processes releasing a lock have to skip over their preempted peers repeatedly. We consider the (unlikely) possibility of starvation insignificant in comparison to this overhead.

4.2 Reader-Writer Locks

Reader-writer locks are a refinement of mutual exclusion locks. They provide exclusive access to a shared data structure on the part of writers (processes making changes to the data), but allow concurrent access by any number of readers. There are several versions of reader-writer locks, distinguished by the policy they use to arbitrate among competing requests from both readers and writers. Reader-preference locks always force writers to wait until there are no readers interested in acquiring the lock. Writer-preference locks always force readers to wait until there are no interested writers. Fair variants prevent newly-arriving readers from joining an active reading session when there are writers waiting, and grant a just-released lock to the process(es) that have been waiting the longest.

Pseudocode for our scheduler-conscious reader-writer lock appears in figures 5 through 7. It is based on a fair scalable reader-writer lock devised by Krieger et al. [19]. When a writer releases a lock for which both readers and writers are waiting, and the longest waiting unpreempted process is a reader, the code grants access to all readers that have been waiting longer than any writers. An alternative interpretation of fairness would grant access in the same situation to *all* currently-waiting unpreempted readers. Like the Smart Queue lock, the reader-writer lock uses both kernel extensions KE-1 and KE-2. It would probably be possible to rewrite the algorithm to use a handshaking protocol and the simpler Symunix interface, but the algorithm is already so complex that we have not attempted to deal with the extra complexity.

Requests for the lock are inserted in a doubly linked list. A reader arriving at the lock checks the status of the previous request. If the previous request is an active reader or if there is no previous request, then the newly-arriving reader marks itself as an active reader and proceeds. In all other cases the newly-arriving process spins, waiting to be released by its predecessor. A process releasing a lock must first remove itself from the queue. If the process is a writer this is an easy task since it has no predecessor in the queue and the procedure is similar to the one followed in the mutual exclusion section. If it is a reader however, then the process may have to remove itself from the middle of the queue. To ensure correct manipulation of the linked list data structure a reader process locks both its own list node and that of its predecessor. It then updates the link pointers to reflect the new state of the list. The locks protecting individual list elements use test_and_set. We have opted for this type of lock because the critical sections are short and the maximum number of contending processes is three. When an unlocking reader attempts, unsuccessfully, to acquire the lock on its predecessor's list element, it re-checks the identity of the predecessor in case it has changed as a result of action on the part of whoever was holding the lock.

After a process has linked itself out of the queue, it must wake up its successor if there is one. The procedure is similar to the one followed in the mutual exclusion case. The releasing process checks the state of its successor and attempts to set its state to unpreemptable_other. If the attempt is successful, the releasing process proceeds to notify

19

```
type rw_qnode = record
    self : ^context_block
    state : (reader, active_reader, writer)
    spin_flag : (waiting, success, failure)
    next, prev : ^rw_qnode
    exc_lock : exclusive_lock
type rw_lock = ^rw_qnode
private cb : ^context_block


procedure writer_lock (L : ^rw_lock, I : ^rw_qnode)
    I->self := cb
    repeat
        cb->state := unpreemptable_self
        I->state := writer
        I->spin_flag := waiting
        I->next := nil
        pred : ^rw_qnode := fetch_and_store (L, I)
        if pred != nil
            pred->next := I
            (void) compare_and_store (&cb->state,
                                      unpreemptable_self, preemptable)
            repeat while I->spin_flag = waiting     // spin
        else
            return
    until I->spin_flag = success


procedure writer_unlock (L: ^rw_lock, I: ^rw_qnode)
    shadow : ^rw_qnode := I
    candidate : ^rw_qnode := I->next
    loop
        if candidate = nil
            if compare_and_store (L, shadow, nil)
                exit loop                           // no one waiting for lock
            repeat while shadow->next = nil         // spin; probably non-local
            candidate := shadow->next
        // order of following checks is important
        if compare_and_store (&candidate->self->state,
                              unpreemptable_self, unpreemptable_other)
              or compare_and_store (&candidate->self->state, preemptable,
                                    unpreemptable_other)
            candidate->prev := nil
            candidate->spin_flag := success
            exit loop
        // else candidate seems to be preempted
        shadow := candidate                         // move down queue
        candidate := shadow->next
        shadow->spin_flag := failure
    cb->state := preemptable
    if cb->warning
        yield
```

Figure 5: Scheduler-conscious reader-writer lock (declarations and writer part).

```
procedure reader_lock (L : ^rw_lock, I : ^rw_qnode)
    I->self := cb
    exc_lock (I)
    repeat
        cb->state := unpreemptable_self
        I->next := I->prev := nil
        I->state := reader
        I->spin_flag := waiting
        pred : ^rw_qnode := fetch_and_store (L, I)
        if pred = nil
            exit loop                // leave repeat
        I->prev := pred
        pred->next := I
        if pred->state = active_reader
            exit loop                // leave repeat
        compare_and_store (&cb->state,
                           unpreemptable_self, preemptable)
        repeat while I->spin_flag = waiting
    until I->spin_flag = success
    I->state := active_reader
    candidate : ^rw_qnode := I->next
    loop
        if candidate = nil or candidate->state != reader
            exit loop
        // order of following checks is important
        if compare_and_store (&candidate->self->state,
                              unpreemptable_self, unpreemptable_other)
               or compare_and_store (&candidate->self->state,
                                     preemptable, unpreemptable_other)
            candidate->spin_flag := success
            exit loop
        // else candidate seems to be preempted
        if candidate->next = nil
            I->next := nil
            if compare_and_store (L, candidate, I)
                // we are now tail of queue
                candidate->spin_flag := failure
                exit loop
            // else need to spin until successor establishes pointers
            repeat while candidate->next = nil
        // preempted candidate has a successor
        I->next := candidate->next
        candidate->next->prev := I
        candidate->spin_flag := failure
        candidate := I->next
    exc_unlock (I)
```

Figure 6: Scheduler-conscious reader-writer lock (reader lock part).

21

```
procedure reader_unlock (L : ^rw_lock, I : ^rw_qnode)
 find_previous:
     pred : ^rw_qnode := I->prev
     if pred = nil  goto no_previous
     while !exc_lock_conditional (pred)
         pred := I->prev
         if pred = nil  goto no_previous
     if pred != I->prev
         exc_unlock (pred)
         goto find_previous
     exc_lock (I)
     pred->next := nil
     if I->next = nil
         if !compare_and_store (L, I, I->prev)
             repeat while I->next = nil                   // spin
     if I->next != nil
         I->next->prev := I->prev
         I->prev->next := I->next
     exc_unlock (pred)
     goto rtn
 no_previous:
     exc_lock (I)
     loop
         candidate : ^rw_qnode := I->next
         if candidate = nil
             if compare_and_store (L, I, nil)  goto rtn
             repeat while I->next = nil                   // spin
         else
             if candidate->self->state = unpreemptable_other
                     or compare_and_store (&candidate->self->state,
                                           unpreemptable_self, unpreemptable_other)
                     or compare_and_store (&candidate->self->state,
                                           preemptable, unpreemptable_other)
                 candidate->prev := nil
                 candidate->spin_flag := success
                 goto rtn
             // else candidate seems to be preempted
             if candidate->next = nil
                 if compare_and_store (L, candidate, nil)
                     // no one at tail of queue
                     candidate->spin_flag := failure
                     goto rtn
                 repeat while candidate->next = nil       // spin
             // preempted candidate has a successor
             I->next := candidate->next
             candidate->next->prev := I
             candidate->spin_flag := failure
 rtn:
     exc_unlock (I)
     cb->state := preemptable
     if cb->warning  yield
```

Figure 7: Scheduler-conscious reader-writer lock (reader unlock part).

its successor that it has been granted the lock. If the attempt fails, it notifies its successor of failure by setting a flag in the successor's node, and proceeds to the next process in the queue. When notified that it has been granted the lock, a reader uses this same procedure to release its own successor, if that successor is also a reader.

4.3 Barriers

We present four scheduler-conscious barriers in this section. The first three are designed for smaller bus-based multiprocessors, or for small partitions on larger machines, in which migration is assumed to be relatively inexpensive. The barriers differ in the amount of information they use in order to make their decisions, and in the quality of those decisions. The first two require no kernel extensions; they employ heuristics and are competitive. The third barrier employs kernel extension KE-3 to make optimal spin versus block decisions. The fourth barrier is designed for large-scale multiprocessors, on which migration is assumed to be an expensive, uncommon event. This barrier makes optimal spin versus block decisions within each processor (or within each cluster of a machine in which migration is inexpensive among small sets of processors), uses a logarithmic-time scalable barrier across processors/clusters, and adapts dynamically to changes in the allocation of processes to processors or processors to applications.

Barrier synchronization's primary source of performance loss in multiprogrammed environments is the cycles wasted spinning while waiting for preempted processes to arrive at the barrier. In order to reduce the performance penalty of wasted spinning, processes can choose to block and relinquish their processor to a preempted peer. Blocking however can be expensive, especially on modern processors, due to the large amount of state that needs to be saved. There are several possible ways to resolve this tradeoff, ranging from always spin to always block. For a dynamically changing environment neither of the extremes provides a satisfactory solution. Inspired by the competitive spinning techniques used by Karlin et al. for locks [16], we have developed a set of competitive techniques for barriers [17]. These techniques choose between spinning and blocking based on the amount of time spent waiting at previous barrier episodes. Since maintaining history is expensive, and since the recent past is generally a better predictor of the future than is the distant past, we base our decisions on the times observed at the last few barrier episodes only.

Two competitive, heuristic-based barriers appear in figures 8 and 9. The first barrier spins for a fixed amount of time and then blocks. By setting the time spent in spinning equal to the context switch time, we can guarantee that the algorithm takes at most twice as long as necessary. The second barrier gathers information from the last three barrier episodes and shortens or lengthens its spinning threshold based on the observed waiting time. Additional heuristics are explored in a previous paper [17].[6]

Competitive spinning barriers provide a simple way to achieve acceptable performance in a multiprogrammed environment. They require no kernel extensions, and work reasonably well if process arrival times are not significantly skewed. They depend on changes to the set of processors available to an application being relatively infrequent, compared to the rate at which barriers are encountered.

---

[6]The pseudocode of the previous paper has been modified slightly here for the sake of consistency.

```
shared global_sense, barrier_count, num_blocked : integer := 0, 0, 0
shared wakeup_sems : array [2] of semaphore := {0}
shared mutex : lock
private local_sense : integer := 0

procedure barrier ()
    local_sense := 1 - local_sense
    count : integer := fetch_and_increment (&barrier_count)
    if count < NUM_PROCESSES - 1
        for i : integer in 1..SWITCH_TIME
            if global_sense = local_sense
                return
        acquire_lock (mutex)
        if global_sense = local_sense
            release_lock (mutex)
            return
        num_blocked +:= 1
        release_lock (mutex)
        P (wakeup_sem[local_sense])
    else
        barrier_count := 0
        acquire_lock (mutex)
        global_sense := 1 - global_sense     // release spinning processes
        count := num_blocked
        num_blocked := 0
        release_lock (mutex)
        for i in 1..count
            V (wakeup_sems[local_sense])     // release blocked processes
```

Figure 8: Fixed time (spin-block) competitive barrier.

24

```
shared global_sense, barrier_count, num_blocked : integer := 0, 0, 0
shared wakeup_sems : array [2] of semaphore := {0}
shared mutex : lock
private local_sense : integer := 0
private spin_threshold : integer := SWITCH_TIME
private episode_count : integer := 0
private episode_time : array [3] of integer := {SWITCH_TIME}

procedure barrier ()
    local_sense := 1 - local_sense
    count : integer := fetch_and_increment (&barrier_count)
    if count < NUM_PROCESSES - 1
        now : integer := get_current_time ()
        for i in 1..spin_threshold
            if global_sense = local_sense
                goto exit_barrier
        acquire_lock (mutex)
        if global_sense = local_sense
            release_lock (mutex)
            goto exit_barrier
        num_blocked +:= 1
        release_lock (mutex)
        P (wakeup_sem[local_sense])
    exit_barrier:
        episode_time[episode_count] := get_current_time () - now
        episode_count := (episode_count + 1) % 3
        if average (episode_time) < SWITCH_TIME
            spin_threshold := min (SWITCH_TIME, spin_threshold + ADJUST)
        else
            spin_threshold := max (0, spin_threshold - ADJUST)
    else
        barrier_count := 0
        acquire_lock (mutex)
        global_sense := 1 - global_sense        // release spinning processes
        count := num_blocked
        num_blocked := 0
        release_lock (mutex)
        for i in 1..count
            V (wakeup_sem[local_sense])          // release blocked processes
```

Figure 9: Average three competitive barrier.

```
shared global_sense, barrier_count : integer := 0, 0
shared wakeup_sems : array [2] of semaphore := {0}
shared partition : ^partition_block
shared barrier_processors : array [2] of integer := {partition->num_processors}
private local_sense : integer := 0

procedure barrier ()
    local_sense := 1 - local_sense
    count : integer := fetch_and_increment (&barrier_count)
    if count + 1 < NUM_PROCESSES
        if count + 1 >= NUM_PROCESSES - barrier_processors[local_sense]
            repeat until global_sense = local_sense      // spin
        else
            P (wakeup_sem[local_sense])
    else
        barrier_count := 0
        barrier_processors[1-local_sense] := partition->num_processors
        global_sense := 1 - global_sense
        for i in 1..(NUM_PROCESSES - barrier_processors[local_sense])
            V (wakeup_sem[local_sense])
```

Figure 10: Scheduler Information barrier.

Competitive spinning barriers have several problems, however. One is the inability to handle skewed arrival times gracefully. When processes arrive at a barrier at very different times, the competitive algorithms block, incorrectly concluding that there are not enough processors to accommodate all the processes. In addition, the competitive algorithms use a uniform policy for all processes: either all will spin or all will block. Ideally on a system with N processes and P processors (and inexpensive migration) the first $N - P$ processes should block while the remaining P should spin. By using kernel extension KE-3, the application can be made aware of the number of available processors. By keeping an internal count of the number of processes already at the barrier and by knowing the number of processors, each individual process can make the optimal choice as whether to spin or to block when arriving at a barrier. This approach has low overhead (a check against the number of available processors) and makes the optimal spin versus block decision. Pseudocode for the *Scheduler Information* barrier appears in figure 10.

Barriers for large-scale multiprocessors are of necessity more complicated since counter based algorithms are too slow (linear in the number of processes) and cause too much contention. Scalable barrier algorithms use log-depth data structures to note the arrival and signal the departure of processes. Unfortunately, these data structures tend to exacerbate the problems caused by multiprogrammed environments, since they require portions of the barrier code in different processes to be interleaved in a deterministic order. This order may conflict with the scheduling policy on a multiprogrammed system, consequently causing an unreasonable number of context switches [28] to occur before achieving the barrier.

The basic idea of our scalable scheduler-conscious barrier is to make the optimal spin versus block decision within each individual processor or cluster, and to employ a scalable log-depth barrier across processors, where context switches are not an issue. Specifically,

26

```
type whole_and_parts = union
    whole : long
    parts: array [4] of byte
type tree_node = record
    have_child : whole_and_parts
    child_not_ready : whole_and_parts := have_child
    parent_flag : ^byte
    dummy : byte                    // something harmless to point at
type processor_info = record
    barrier_count : integer := 0
    wakeup_sems : array [2] of semaphore := {0}
    generation : integer := 0       // used to synchronize reorganization
shared processors : array [MAX_PROCESSORS] of processor_info
shared nodes : array [MAX_PROCESSORS] of tree_node
    // have_child and parent_flag fields of individual nodes are initialized
    // as appropriate in the inter-processor tree; see code in reorganize ()
shared global_sense : integer := 0
shared partition : ^partition_block
shared barrier_partition : partition_block := partition^
private local_sense : integer := 0
private cb : ^context_block
private process_id : integer :=     // unique number in 0..NUM_PROCESSES-1
private my_processor : integer := partition->processor_ids[process_id]
private my_generation : integer := 0

procedure barrier ()
    local_sense := 1 - local_sense
    L : processor_info := &processors[my_processor]
    count : integer := fetch_and_increment (&L->barrier_count)
    if count + 1 < barrier_partition.processes_on_processor[my_processor]
        // not the last process on the processor
        P (L->wakeup_sems[local_sense])
        goto rtn
    // last process on this processor; wait for children on other processors
    my_node : ^tree_node := &nodes[my_processor]
    repeat while my_node->child_not_ready.whole <> 0     // spin
    // barrier has been achieved
    my_node->child_not_ready.whole := my_node->have_child.whole
    my_node->parent_flag^ := 0                   // notify parent
    if my_processor = 0                          // root of inter-processor tree
        // copy partition information if necessary; loop ensures atomicity
        check : integer := barrier_partition.generation
        while check <> partition->generation
            check := partition->generation
            barrier_partition := partition^
        global_sense := local_sense          // release spinning processes
    else repeat while global_sense != local_sense        // spin
    L->barrier_count := 0                         // reset for this processor only
    for i in 1..count
        V (L->wakeup_sems[local_sense])       // release blocked processes
rtn:
    if my_generation != barrier_partition.generation  reorganize ()
```

Figure 11: Scheduler-conscious tree barrier.

```
procedure reorganize ()
    my_generation := barrier_partition.generation
    my_processor := barrier_partition.processor_ids[process_id]
    my_node : ^tree_node := &nodes[my_processor]
    for i in 0..process_id-1
        if barrier_partition.processor_ids[i] = my_processor
            // I'm not the representative of my processor
            repeat until processors[my_processor].generation = my_generation
                // spin
            return
    for i in 0..3
        my_node->havechild.parts[i] :=
            (integer) ((my_processor*4 + i+1) < barrier_partition.num_processors)
    my_node->childnotready.whole := my_node->havechild.whole
    if my_processor = 0                    // root of inter-processor tree
        my_node->parentflag := &my_node->dummy
        processors[my_processor].generation := my_generation
            // signal children it is safe to proceed
    else
        parent_id : integer := (my_processor-1)/4
        my_node->parentflag :=
            &nodes[parent_id].childnotready.parts[(my_processor-1)%4]
        processors[my_processor].generation := my_generation
        repeat until processors[parent_id].generation = my_generation
            // spin
```

Figure 12: Scheduler-conscious tree barrier reorganization.

we combine the Scheduler Information barrier of figure 10 with the scalable tree barrier of Mellor-Crummey and Scott [30]. Processes assigned to the same processor or cluster use a Scheduler Information barrier. The last process to reach the barrier becomes the representative process for the processor/cluster. Representative processes participate in the tree barrier.

A partition generation count allows us to handle repartitioning—changes in the mapping of processes to processors or clusters. We shadow this generation count with a count that belongs to the barrier. The process at the root of the inter-processor barrier checks the barrier generation count against the partition generation count. If the two counts are found to be different, processes within each new processor/cluster elect a representative and the representatives then go through a barrier reorganization phase, initializing tree pointers appropriately.[7] This approach has the property that barrier data structures can be reorganized only at a barrier departure point. As a result, processes may go through one episode of the barrier using outdated information. While this does not affect correctness it could have an impact on performance. If repartitioning were a frequent event, then processes would use old information too often and performance would suffer. However, we consider it unlikely that repartitioning would occur more than a few times per second on a large-scale, high-performance machine, in which case the impact of using out-of-date barrier data structures would be negligible. Pseudocode for the scheduler-conscious scalable barrier appears in figures 11 and 12.

5 Results

This section presents a performance evaluation of different scheduler-conscious synchronization algorithms, including a comparison to the best known scheduler-oblivious algorithms. We begin by describing our experimental methodology. We then consider mutual exclusion, reader-writer locks, and barriers in turn.

5.1 Methodology

We have tested our algorithms on two different architectures. As an example of a small-scale bus-based machine we use a 12-processor Silicon Graphics Challenge. As an example of a large-scale distributed memory machine we use a 64-processor partition of a Kendall Square KSR 1. We have used both synthetic and real applications. The synthetic applications allow us to thoroughly explore the parameters that may affect synchronization performance, including the ratio between the lengths of critical and non-critical sections, the degree of multiprogramming, the quantum size, and others. The real applications allow us to validate our findings in the context of a larger computation, potentially capturing effects that are missing in the synthetic applications, and providing a measure of the impact of the synchronization algorithms on overall system performance.

---

[7]Note that the representative for the re-organization phase is not necessarily the process that will participate in the inter-processor phase of subsequent barriers; this latter role is played by the last process to arrive at each individual intra-processor barrier.

Our synchronization algorithms employ atomic operations not available on either of the two target architectures. We have implemented software versions of these instructions using small critical sections bracketed by the native synchronization primitives of the machines (test_and_set implemented by the load_linked and store_conditional instructions on the Challenge, get_subpage (lock cache line) and free_subpage (unlock cache line) on the KSR). While this approach adds overhead to the algorithms, the overhead is small.[8] Moreover, because we are running scalable algorithms, in which processes use backoff or spin only on local locations, competition is essentially non-existent for the critical sections that implement the "atomic" operations, and does not result in any significant increase in overall levels of network and memory contention. Our results for the non-native locks are therefore slightly higher in absolute time, but qualitatively very close in character, to what would be achieved with hardware supported fetch_and_Φ instructions.

For the sake of simplicity, we employ a user-level scheduler in our experiments. One processor is dedicated to running the scheduler. While the kernel interface described in section 3 would not be hard to implement, it was not needed for our experiments (we also lacked the authorization to make kernel changes on the KSR machine). For the lock experiments, each application process has its own processor. Preemption is simulated by sending a user-defined signal to the process. The process catches this signal and jumps to a handler where it spins on a flag that the scheduler process will set when it is time to return to executing application code. The time spent spinning is meant to represent execution by one or more processes belonging to other, unrelated applications.

For the scalable barriers, multiprogramming is simulated by multiplexing one or more application processes on the same processor. Both the SGI and KSR operating systems provide us with this capability by allowing us to bind processes to processors. The centralized barrier experiments require process migration. On the SGI we can restrict processors (prevent processes from executing on them). Restricting a processor increases the multiprogramming level on the remaining processors. Processes are allowed to migrate among the unrestricted processors. The KSR operating system does not provide an analogue of the SGI restrict operation, so we were unable to control the number of processors available to migrating processes. For this reason we do not report results for the centralized barriers on the KSR.

The *multiprogramming level* reported in the experiments indicates the average number of processes per processor. For the lock-based experiments, one of these processes belongs to our application program; the others are assumed to belong to other applications, and are simulated by spinning in a signal handler as described above. For the barrier-based experiments, multiple application processes reside on each processor, and participate in all the barriers. The reason for the difference in methodology is that for lock-based applications we are principally concerned about processes being preempted while holding a critical resource, while for barrier-based applications we are principally concerned about processes wasting processor resources while their peers could be doing useful work. Our lock algorithms are designed to work in any multiprogrammed environment; the scalable barrier assumes that

---

[8]On the SGI machine we we could have used load_linked and store_conditional directly on the atomic variable, rather than on an associated flag, but the difference in code lengths is insignificant, and the added uniformity between machines made the experiments slightly easier.

processors are partitioned among applications. A multiprogramming level of 1 indicates one application process on each processor. Fractional multiprogramming levels indicate additional processes on some, but not all, processors.

5.2 Mutual Exclusion

We implemented ten different mutual exclusion algorithms, covering scheduler-conscious and scheduler-oblivious, and scalable and centralized locks:

TAS-B – A standard test-and-test_and_set lock with bounded exponential backoff. This algorithm repeatedly reads a central flag until it appears to be unset, then attempts to set it atomically in order to acquire the lock. On the SGI Challenge, this is the native lock, augmented with backoff.

TAS-B-np – The same as TAS-B, but avoids preemption in critical sections by using the Symunix kernel interface.

Queue – A list-based queued lock with local-only spinning [30].

Queue-np – An extension to the Queue lock that avoids preemption in critical sections, also using the Symunix kernel interface. This algorithm does *not* avoid passing the lock to a process that has been preempted while waiting in line.

Queue-HS – An extension to the Queue-np lock that uses handshaking to ensure that the lock is not transferred to a preempted process. This algorithm appears in figure 2.

Smart-Q – An alternative extension to the Queue-np lock that uses kernel extensions KE-1 and KE-2 to obtain simpler code and lower overhead than in the Queue-HS lock. This algorithm appears in figure 3.

Ticket – The standard ticket lock with proportional backoff, but with no special handling of preemption in the critical section or the queue.

Ticket-np – A scheduler-conscious ticket lock with backoff, using handshaking to avoid preemption in the critical section. This algorithm appears in figure 4.

Native – A lock employing machine-specific hardware. This is the standard lock that would be used by a programmer familiar with the machine's capabilities.

Native-np – An extension to the native lock that uses the Symunix kernel interface to avoid preemption while in the critical section.

The Native lock on the SGI Challenge is a test-and-test_and_set lock implemented using the load_linked and store_conditional instructions of the R4400 microprocessor. The Native lock on the KSR 1 employs a cache line locking mechanism that provides the equivalent of queued locks in hardware. The queuing is based on physical proximity in a ring-based interconnection network, rather than on the chronological order of requests. We would expect the Native-np locks to outperform all other options on these two machines,
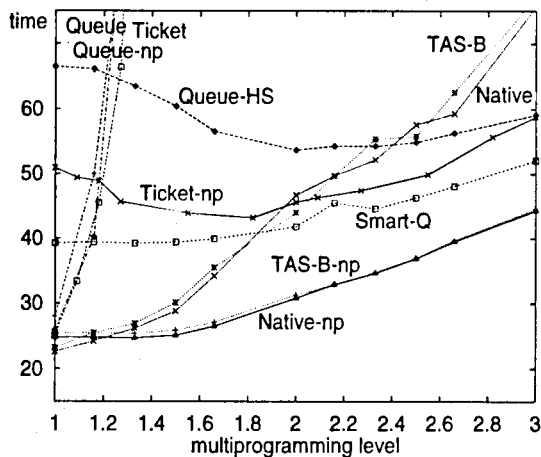
31

Figure 13: Varying multiprogramming level on an 11-processor SGI Challenge.
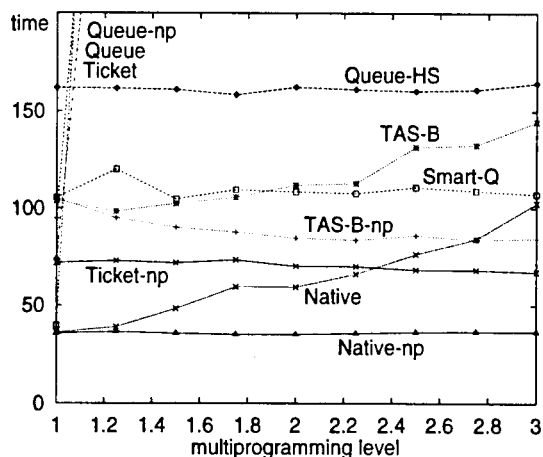

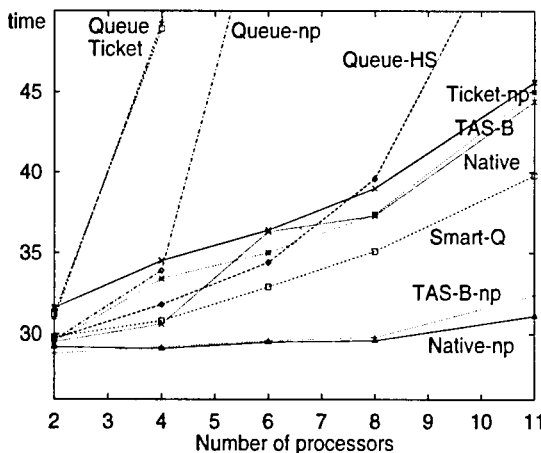Figure 14: Varying multiprogramming level on a 63-processor KSR 1.

Figure 15: Varying the number of processors on the SGI Challenge with a multiprogramming level of 2.
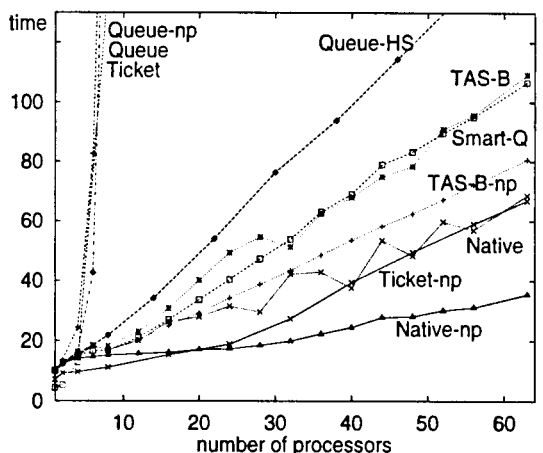

Figure 16: Varying the number of processors on the KSR 1 with a multiprogramming level of 2.

not only because they make use of special hardware, but because the atomic operations in all the other locks are built on top of them. Our experiments confirm this expectation.

Our synthetic application executes a simple loop consisting of a work section and a critical section. The total number of loop iterations is proportional to the number of executing processes. In designing a scalable synthetic application we needed to ensure that the critical section did not become a bottleneck. Therefore, we set the ratio of the lengths of the critical and non-critical sections on both machines to slightly less than the inverse of the maximum number of processors. Absolute quantum length (in cycles or microseconds) had no significant effect on performance. We therefore concentrate here on the remaining variables in the synthetic application: multiprogramming level and number of processors.

Figures 13 and 14 plot execution time of the synthetic application against multiprogramming level for a fixed number of processors (11 on the SGI and 63 on the KSR). On the SGI, the scheduling quantum is fixed at 20 ms and the critical to non-critical section ratio is 1:14. We used a random number generator to vary the length of the critical section within a narrow range, to more closely approximate the behavior of real applications and to avoid possible lock-stepping of the different processes. The Queue, Queue-np, and Ticket locks show the worst degradation, because processes queue up behind preempted peers. Preventing preemption in the critical section helps a little, but not much: preemption of processes waiting in the queue is the dominant problem.

Considerably better behavior is obtained by preventing critical section preemption *and* ensuring that the lock is not given to a blocked process waiting in the queue: the Queue-HS, Smart-Q, and Ticket-np locks perform far better than the other scalable locks, and also outperform the Native and TAS-B locks at multiprogramming levels of 2 and above. The Native-np and TAS-B-np locks display the best results, though past work [30] suggests that they will generate more bus traffic than the scalable locks, and would interfere more with the data-access memory traffic of a real program.[9]

On the KSR, the scheduling quantum is fixed at 50 ms and the ratio of critical to non-critical section lengths is 1:65. The results show slightly different behavior from that on the SGI. The Queue, Queue-np, and Ticket locks suffer an even greater performance hit as the multiprogramming level increases. The Queue-HS lock improves performance considerably, since it eliminates both the critical section and queue preemption problems. Unfortunately, it requires a significant number of high-latency remote references, resulting in a high, steady level of overhead. The Smart-Q lock lowers this level by a third, but it is still a little slower than the TAS-B-np lock. The best non-native lock is Ticket-np. This is expected behavior, since the Ticket lock deals well with contention (not as well as the queued lock, but there is less contention observed in a multiprogrammed environment), and Ticket-np has solved the preemption problem.

The Native-np lock provides the best overall performance. Since all the non-native locks use native locks internally to implement atomic operations, this too is expected behavior. The TAS-B and Native locks perform well when the multiprogramming level is low, but deteriorate as it increases. If the necessary atomic operations (`fetch_and_add`, `swap`, etc.)

---

[9]The synthetic application does not capture this effect; it operates almost entirely out of registers during its critical and non-critical sections. The impact on data-access traffic can be seen in our experiments with real applications.
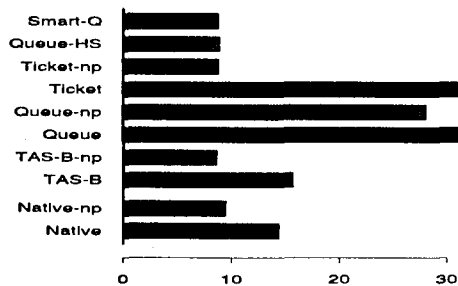
Figure 17: Completion time (in seconds) for Cholesky on the SGI.
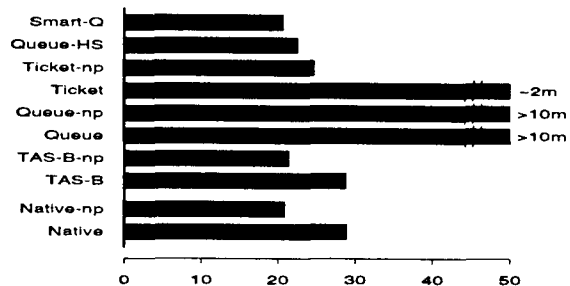

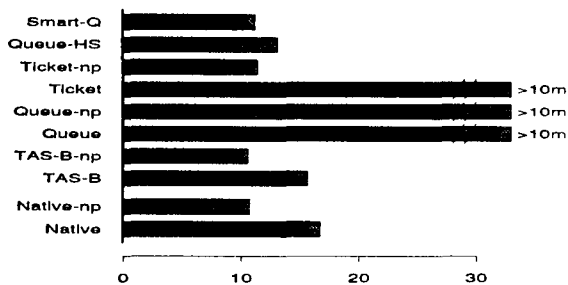Figure 18: Completion time (in seconds) for Cholesky on the KSR.

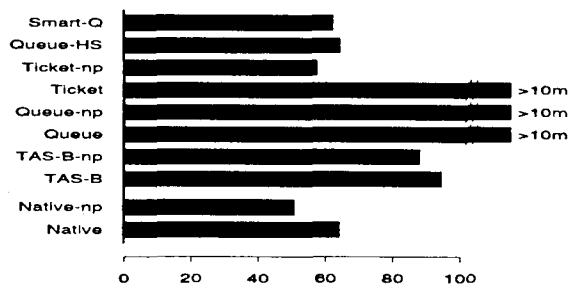Figure 19: Completion time (in seconds) for Quicksort on the SGI.

Figure 20: Completion time (in seconds) for Quicksort on the KSR.

were available on the KSR 1, we would expect the queued and ticket locks to perform better than they do by a small constant factor. The closeness with which those locks follow the performance of KSR's relatively complex built-in primitive suggests that that primitive is probably not cost effective.

Increasing the number of processors working in parallel can result in a significant amount of contention, especially if the program needs to synchronize frequently. Previous work has shown that queued locks improve performance in such an environment, but as indicated by the graphs in figures 13 and 14 they experience difficulties under multiprogramming. The graphs in figures 15 and 16 show the effect of increasing the number of processors on the different locks at a multiprogramming level of 2.

The synthetic program runs a total number of loop iterations proportional to the number of processors, so execution time should not decrease as processors are added. Ideally, it would remain constant, but contention and scheduler interference will cause it to increase.

With quantum size and critical to non-critical ratio fixed as before, results on the SGI again show the Queue, Queue-np, and Ticket locks performing poorly, as a result of untimely preemption. The performance of the TAS-B and Native locks also degrades with additional processors, because of increased contention and because of the increased likelihood of preemption in the critical section. The Smart-Q and Ticket-np locks degrade more slowly, but also appear to experience higher overheads. Increasing the number of processors does not affect the TAS-B-np and Native-np locks until there are more than about eight processors active (the point at which bus contention becomes an issue).

The results on the KSR indicate that contention effects are important for larger numbers of processors. The native lock, with our modification to avoid critical section preemption, is roughly twice as fast as the nearest competition, because of the hardware queuing effect. Among the all-software locks, Ticket-np performs the best but TAS-B-np and Smart-Q are still reasonably close.

Backoff constants for the TAS-B and Ticket locks were determined by trial and error. The best values differ from machine to machine, and even from program to program. The queued locks are more portable. As noted above, contention on both machines becomes a serious problem sooner if the code in the critical and non-critical sections generates memory traffic. As witnessed from real application results, the queued locks suffer less from this effect.

To verify the results obtained from the synthetic program, and to investigate the effect of memory traffic generated by data accesses, we measured the performance of a pair of real applications: the Cholesky program from the Stanford SPLASH suite [39], and a multiprocessor version of Quicksort. These programs contain no barriers; they synchronize only with locks. Figures 17 to 20 show their completion times, in seconds, when run with a multiprogramming level of 2 using 11 processors on the SGI and 63 processors on the KSR. As with the synthetic program, scheduler-oblivious queuing of preemptable processes is disastrous. This time, however, with real computation going on, the Ticket-np and Smart-Q locks match the performance of the TAS-B-np and Native-np locks on the SGI, and to outperform TAS-B-np in the Quicksort program on the KSR.

5.3   Reader-Writer Locks

We implemented six different reader-writer locks:

RW-TAS-B – A centralized reader-writer lock based on a standard test-and-`test_and_set` lock with exponential backoff.

RW-TAS-B-np – The same as RW-TAS-B, but with avoidance of preemption in critical sections, using the Symunix kernel interface.

RW-Queue – A scalable reader-writer lock based on the lock by Krieger et al. [19].

RW-Smart-Q – An extension to the RW-Queue lock that uses kernel extensions KE-1 and KE-2 to avoid preemption in the critical section, and to avoid passing the lock to a preempted process. This algorithm appears in figures 5 through 7.
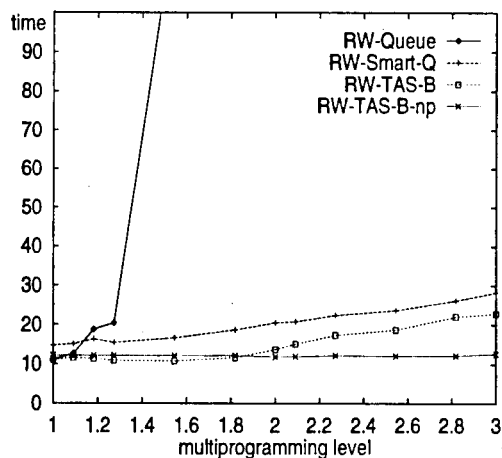
35

Figure 21: Varying the multiprogramming level for the reader-writer lock on the SGI (11 processors).
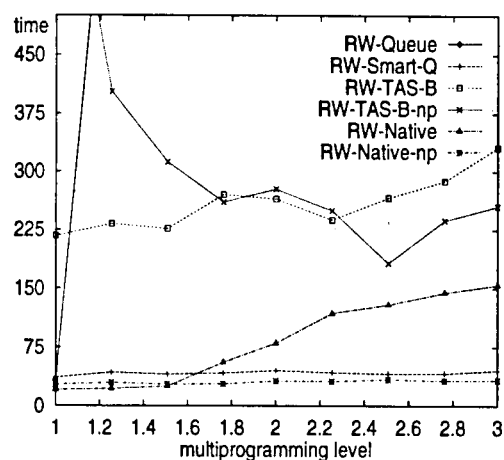

Figure 22: Varying the multiprogramming level for the reader-writer lock on the KSR (63 processors).
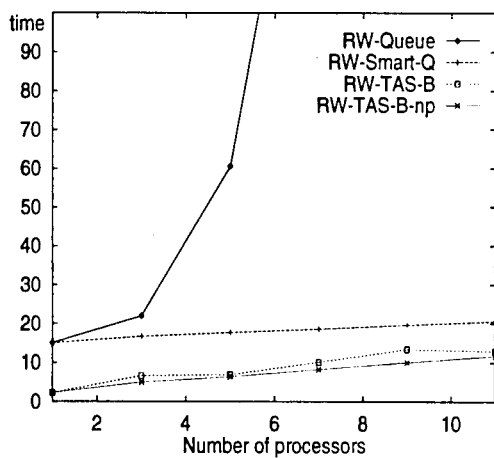

Figure 23: Varying the number of processors for the reader-writer lock on the SGI (multiprogramming level = 2).
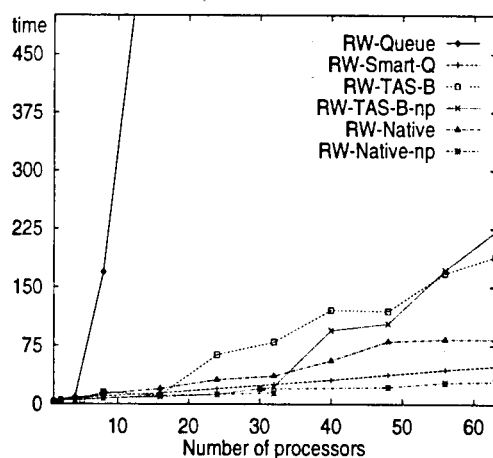

Figure 24: Varying the number of processors for the reader-writer lock on the KSR (multiprogramming level = 2).

RW-Native – A reader-writer lock based on the native synchronization primitive. On the SGI this is identical to the RW-TAS-B lock.

RW-Native-np – The same as RW-Native, but with avoidance of preemption in critical sections, using the Symunix kernel interface. On the SGI this is identical to the RW-TAS-B-np lock.

Figures 21 and 22 show the performance of the various reader-writer locks under varying levels of multiprogramming on the SGI (11 processors) and KSR (63 processors), respectively. Figures 23 and 24 show performance on varying numbers of processors, at a multiprogramming level of 2.

Reader-writer locks display behavior similar to that of mutual exclusion locks. The RW-Native-np lock outperforms all the others in a multiprogrammed environment. The RW-Smart-Q lock is a close second. The algorithms that do not cope with preemption behave increasingly worse as the multiprogramming level increases, though this effect is less pronounced than it was in the case of mutual exclusion. Five percent of the critical sections in our experiments acquire a writer lock; the rest acquire a reader lock, and can proceed in parallel with other readers. Preempting a process that holds a lock usually means preempting a reader, not a writer, so other readers can still proceed (so long as a writer is not yet in line).

As in the case of mutual exclusion, the centralized RW-TAS-B and RW-TAS-B-np locks still suffer from contention on large numbers of processors. The contention effects observed with reader-writer locks are more pronounced than what was observed with the mutual exclusion locks. In contrast to those earlier experiments, we did not try to maintain a ratio of regular work and critical section work inversely proportional to the number of processors. We assumed that the additional parallelism available due to the concurrency of readers would reduce the observed contention. This turned out not to be the case: reader-writer locks experienced a high degree of contention. Graph 22 shows that the centralized version of the lock based on the `test_and_set` primitive actually improves in performance as multiprogramming increases. The reason is that the increase in multiprogramming reduces the contention experienced by processes and allows lock operations to complete faster, even though there are fewer processor cycles available to the application.

For completeness, we ran experiments with one, five, and fifty percent writers. Larger numbers of writers cause a higher degree of contention—expected since there is less concurrency available—and degrade the performance of the RW-TAS-B and RW-TAS-B-np locks. We present the five percent results here. The others are qualitatively similar.

5.4 Barriers

We present results on barrier synchronization in two sections, one for small-scale machines such as the SGI Challenge (these results also apply to small partitions of a larger machine), and one for large-scale machines such as the KSR 1.

5.4.1 Small-scale barriers

For small-scale, centralized barriers, we implemented three baseline cases—always spin, always block, and spin-then-block—, three competitive algorithms that adjust their behavior based on previous barrier episodes, and the scheduler information barrier of figure 10, which uses kernel extension KE-3 to make an optimal spin versus block decision:

C-spin – All processes spin while waiting for their peers to reach the barrier.

C-block – Processes never spin; if they need to wait, they place themselves on a semaphore queue. The last process to arrive at the barrier wakes up its peers by performing V operations on the semaphore.

C-sp-blk – Processes spin for a bounded amount of time equal to the cost of a context switch. If the bound expires before the barrier is achieved, then the process yields the processor by performing a P operation on a semaphore. The last process to arrive at the barrier checks the semaphore queue and wakes up any processes that are blocked. This algorithm appears in figure 8.

C-last1 – Processes spin for a period of time determined by how long they waited at the previous barrier episode. This time is increased if the process did not exceed it during the last barrier episode and decreased otherwise. The upper bound on the time is the cost of a context switch.

C-avg3 – This barrier is similar to the C-last1 barrier, except that the last three barrier episodes are used in determining the amount of time spent spinning. This algorithm appears in figure 9.

C-coarse – A competitive barrier similar to the C-last1 barrier, except that the spinning time is not adjusted incrementally, but rather all at once.

C-sched – A barrier that makes an optimal spin versus block decision based on the number of available processors (as reported by kernel extension KE-3) and the number of processes that have yet to reach the barrier. This algorithm appears in figure 10.

Figure 25 shows the performance of the synthetic application on the SGI Challenge when using different barrier implementations, as the multiprogramming level increases. Our experiments were run in a dynamic hardware partitioned environment, where the number of processors available to the application varies between 5 and 11, with an average of 7.9. The multiprogramming level is calculated based on the average number of processors and the number of processes used by the application. The synthetic application performs no real work between barriers; it does not capture the effect of data-access memory references. The kernel scheduler moves processes among processors in order to balance load. Processes running on the same processor are multiprogrammed with a quantum length of 30 ms, the default value used by the IRIX kernel scheduler.

In the absence of multiprogramming, the C-sched barrier performs as well as the C-spin barrier—its overhead is low—, and significantly better than the competitive barriers. As
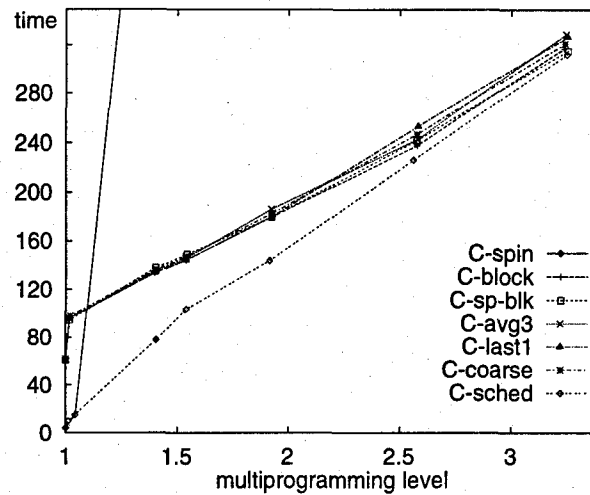
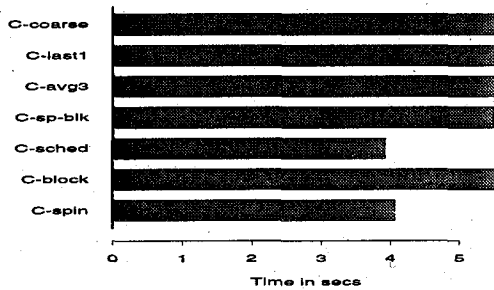Figure 25: Performance of the small-scale barriers for the synthetic program.

Figure 26: Gaussian Elimination run-time for different barrier implementations (multiprogramming level=1).
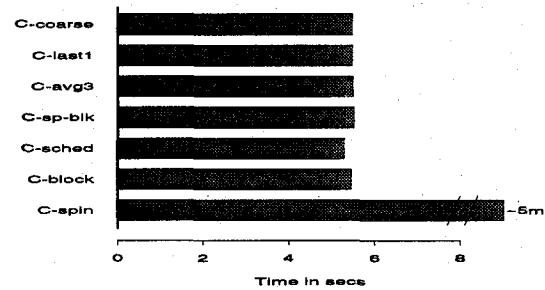

Figure 27: Gaussian Elimination run-time for different barrier implementations (multiprogramming level=2).

the multiprogramming level increases the spinning barrier's performance degrades sharply, while the C-sched barrier retains its good performance and its advantage over the other algorithms. It never spins when other processes could make use of the current processor, and it avoids the overhead of blocking in the last P processes to arrive at the barrier. At very high multiprogramming levels, the Scheduler Information barrier is only slightly faster than the competitive and blocking barriers: as the number of processes per processor increases it becomes less important to avoid blocking in the final process on each processor.

To validate the results obtained with the synthetic application, we experimented with a real application as well: Gaussian elimination. We ran 11 processes on 5–11 processors, with a quantum length of 30 ms and a repartition operation (a change in the number of processors) every 80 ms.

The main difference we observed with respect to the synthetic results is a decrease in the impact of synchronization on overall performance, since it is combined with the time spent
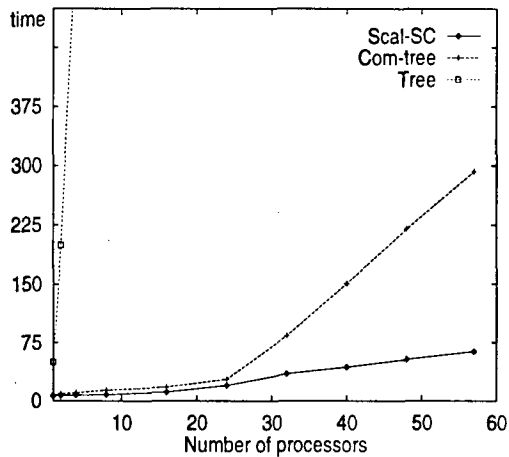
39

Figure 28: Varying the number of processors for the barriers on the KSR (multiprogramming level = 2).
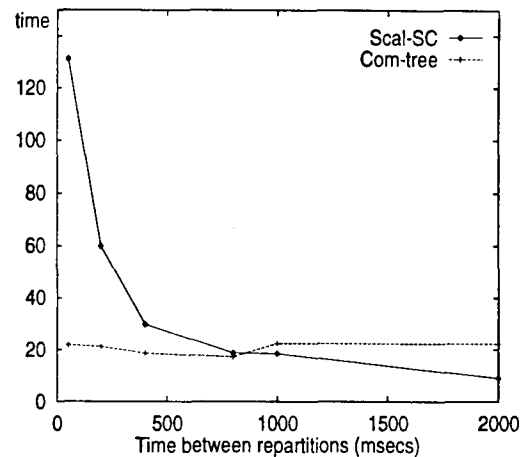
Figure 29: Varying the frequency of repartitioning decisions for the barriers on the KSR (57 processors).

in real computation. Figures 26 and 27 show the completion time of Gaussian elimination at multiprogramming levels of 1 and 2 respectively. In both cases the application receives an average of eight processors during its lifetime, but in the second case there are more (16) application processes cooperating on the problem. In both cases the C-sched barrier provides the best performance.

As with mutual exclusion and reader-writer locks, we experimented with a variety of other values for each of the experimental parameters. The only parameter (other than multiprogramming level and number of processors) to display a noticeable impact on performance was the frequency of repartitioning decisions. As the time between repartitions increases, the performance of the competitive barriers improves to some extent, since they need time to adapt to a change in partition size, and an increase in the time between repartitions allows them to amortize their adaptation cost over a larger number of episodes. The performance of the blocking and spinning barriers is essentially independent of the time between repartitions.

We were initially surprised to see a small but steady improvement in the performance of the C-sched barrier as the time between repartitions increased. The explanation is that it is possible for the algorithm to err when a repartitioning decision occurs at the same time that the application is going through a barrier. In this case, some threads will use old information to guide their decision and thus may decide sub-optimally. When the time between scheduling decisions is large, sub-optimal decisions happen less frequently, resulting in a small performance improvement.

5.4.2 Scalable barriers

For large scale machines we implemented and tested three barriers:
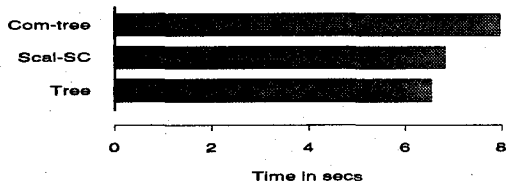
Figure 30: Gaussian elimination run-time on the KSR using 57 processors (multiprogramming level = 1)
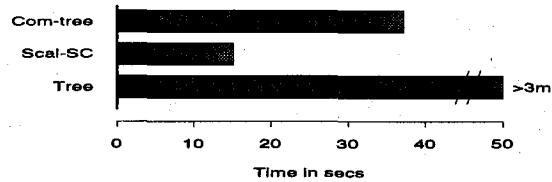
Figure 31: Gaussian elimination run-time on the KSR using 57 processors (multiprogramming level = 2)

Tree – Mellor-Crummey and Scott's tree barrier with flag wakeup [30]. This algorithm associates processes with nodes (both internal and leaves) in a static 4-ary fan-in tree. After waiting for their children (if any) and signaling their parent (if any) in the arrival tree, processes spin on locally-cached copies of a single, global wakeup flag. The last arriving process sets this flag. On KSR's ring-based topology, the resulting invalidations and reloads approximate hardware broadcast.

Com-tree – A competitive variant of the Tree barrier, in which processes spin for only a bounded amount of time, as in the C-sp-blk algorithm of the previous section.

Scal-SC – A scalable scheduler-conscious barrier that uses the C-sched barrier among the processes on a given processor and a scalable tree barrier across processors. Code for this algorithm appears in figures 11 and 12. The code actually run on the KSR differs from the figures in that a process simply yields the processor, rather than blocking on a semaphore, when it discovers that it is not the last arriving process on its processor. If processes can do their inter-barrier work in any order, and if the scheduler does not re-run a process until its peers have had a chance to run, then yielding results in no more context switches than blocking, and saves the last arriving process on each processor from having to do a series of V operations. For correctness, the `yield` calls are embedded in a loop that re-tests `global_sense`.

Barriers based on a centralized counter do not scale well to larger machines for two reasons. First, their critical path length is linear in the number of processors; second, the centralized counter can become a significant source of contention. Given that processes do not migrate among processors, the Scal-SC algorithm avoids these problems while making optimal spin versus block decisions.

Figure 28 compares the performance of the various barriers in our synthetic application on the KSR 1, with a multiprogramming level of two, and with varying numbers of processors. Repartitioning decisions were made at one-second intervals. As can be seen from the graph, the Tree barrier is rendered useless with the introduction of multiprogramming. Its performance degrades due to the large number of context switches required in order to go through a barrier episode, and the amount of time wasted before each context switch — equal to the process' quantum. It is surprising to see that even the "spin then block" heuristic of the Com-tree barrier performs quite badly in the presence of multiprogramming.

41

While processes do not have to waste a quantum before yielding their processors they still have to suffer the large number of context switches that degrade performance. The Scal-SC barrier improves performance by an order of magnitude compared to its closest competitor the Com-tree barrier. It requires the minimum number of context-switches necessary, while still maintaining the logarithmic path length and low contention properties of the Tree barrier.

As we mentioned in section 4, the Scal-SC barrier can be sensitive to the frequency of scheduling decisions. We ran experiments to determine the level of sensitivity. Figure 29 shows that if the time between repartition decisions is very small, performance degrades quite sharply. We believe, however, that repartitioning will be a rare event on large machines—as rare as the arrival and departure of jobs from the system. For repartition intervals greater than 500 ms, the Scal-SC barrier performs well.

To validate the synthetic results, we ran a barrier-based version of Gaussian elimination on 57 KSR processors.[10] The results appear in figures 30 and 31. In the absence of multiprogramming the Scal-SC barrier is only slightly worse than the Tree barrier, and significantly better than the Com-tree barrier. The introduction of multiprogramming renders the Tree barrier useless; its performance degrades by at least an order of magnitude. At the same time, the Scal-SC barrier outperforms Com-tree by more than 50%.

6   Conclusions

In this paper we presented solutions to the problem of synchronization on multiprogrammed multiprocessors, for both small and large-scale machines. We identified the main sources of performance loss for the two most common types of synchronization algorithms: locks and barriers. We also demonstrated that the scalable versions of synchronization algorithms are particularly sensitive to multiprogramming. We then proceeded to define an extended kernel interface allowing communication of process state information between the user and the kernel and we used this interface to construct scheduler-conscious versions of several synchronization algorithms. We demonstrated that these algorithms perform well in the absence of multiprogramming and provide significant performance advantages in a multiprogrammed environment.

We have also found that increasing the multiprogramming level decreases the contention observed by the application since the number of processes accessing a synchronization variable concurrently is reduced. As a result, the scalable synchronization algorithms are sometimes inferior to centralized ones in multiprogrammed environments. Future increases in machine size are likely to increase the number of contending processors in multiprogrammed environments, making scalable synchronization algorithms more desirable. Moreover, it is likely that coherence protocols on future machines will lack the ability to efficiently keep track of a large number of processors sharing a common variable. As a result, the cost of coherence management for the data structures of centralized synchronization algorithms is likely to be unacceptably high.

---

[10]We used pthreads to express parallelism in our barrier experiments. Due to limitations in the pthreads environment on the KSR only 57 of the 64 processors in the partition could be utilized.

Further extensions to the kernel-user interface may allow even greater performance gains to be achieved. Such extensions might include allowing the kernel to run user-supplied functions in response to particular kernel events, or choosing the partition size based on the application's characteristics. For example, it does not make sense to remove a single processor from a 64-processor barrier-based application: the application would run almost as fast on 32 processors. We believe that as large-scale multiprocessors become more common they will inevitably be multiprogrammed, and the importance of exchanging information across the kernel-user boundary will increase.

Acknowledgements

References

[1] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. Originally presented at the *Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[3] N. S. Arenstorf and H. Jordan. Comparing Barrier Algorityhms. *Parallel Computing*, 12:157–170, 1989.

[4] T. S. Axelrod. Effects of Synchronization Barriers on Multiprocessor Performance. *Parallel Computing*, 3:129–140, 1986.

[5] D. L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *Computer*, 23(5):35–43, May 1990.

[6] P. Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[7] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, December 1991.

[8] J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers*, Pittsburgh, PA, September 1988.

[9] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multi-processors. *Computer*, 23(6):60–69, June 1990.

[10] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 120–132, San Diego, CA, May 1991.

[11] D. Hensgen, R. Finkel, and U. Manber. Two Algorithms for Barrier Synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[12] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[13] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[14] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. Scalable Coherent Interface. *Computer*, 23(6):74–77, June 1990.

[15] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[16] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 41–55, Pacific Grove, CA, October 1991.

[17] L. Kontothanassis and R. Wisniewski. Using Scheduler Information to Achieve Optimal Barrier Synchronization Performance. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, May 1993.

[18] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B. Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[19] O. Krieger, M. Stumm, and R. Unrau. A Fair Fast Scalable Reader-Writer Lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[20] C. A. Lee. Barrier Synchronization over Multistage Interconnection Networks. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 130–133, Dallas, TX, December 1990.

[21] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[22] S. T. Leutenegger and M. K. Vernon. Performance of Multiprogrammed Multiprocessor Scheduling Algorithms. In *Proceedings of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990.

[23] B. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, San Jose, CA, October 1994.

[24] B. G. Lindsay and others. Computation and Communication in R*: A Distributed Database Manager. *ACM Transactions on Computer Systems*, 2(1):24–28, February 1984.

[25] B. Lubachevsky. Synchronization Barrier and Related Tools for Shared Memory Parallel Programming. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II:175–179, August 1989.

[26] P. Magnussen, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[27] E. P. Markatos. Multiprocessor Synchronization Primitives with Priorities. In *Proceedings of the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 1–7, Atlanta, GA, May 1991. Held in conjunction with the *Seventeenth IFAC/IFIP Workshop on Real-Time Programming*, and published in the *Newsletter of the IEEE Computer Society Technical Committee on Real-Time Systems* 7:4 (Fall 1991).

[28] E. Markatos, M. Crovella, P. Das, C. Dubnicki, and T. LeBlanc. The Effects of Multiprogramming on Barrier Synchronization. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 662–669, December 1991.

[29] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-Class User-Level Threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110–121, Pacific Grove, CA, October 1991.

[30] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[31] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 106–113, Williamsburg, VA, April 1991.

[32] J. M. Mellor-Crummey and M. L. Scott. Synchronization Without Contention. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, Santa Clara, CA, April 1991.

[33] M. Noakes, D. Wallach, and W. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[34] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: An Experiment in Distributed Operating System Structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[35] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, October 1982.

[36] C. D. Polychronopoulos and D. J. Kuck. Guided Self-scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[37] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Multi-Model Parallel Programming in Psyche. In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 70–78, Seattle, WA, March 1990.

[38] M. L. Scott and J. M. Mellor-Crummey. Fast, Contention-Free Combining Tree Barriers. *International Journal of Parallel Programming*, 22(4):449–481, August 1994. Earlier version available as TR 429, Computer Science Department, University of Rochester, June 1992.

[39] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44, March 1992.

[40] J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[41] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 159–166, Litchfield Park, AZ, December 1989.

[42] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[43] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott. Scalable Spin Locks for Multiprogrammed Systems. In *Proceedings of the Eighth International Parallel Processing*

Symposium, pages 583–589, Cancun, Mexico, April 1994. Earlier but expanded version available as TR 454, Computer Science Department, University of Rochester, April 1993.

[44] H. Yang and J. H. Anderson. Fast, Scalable Synchronization with Minimal Hardware Support (extended abstract). In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, August 1993.

[45] P. Yew, N. Tzeng, and D. H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[46] J. Zahorjan, E. D. Lazowska, and D. L. Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. In *Proceedings of the International Symposium on Performance of Distributed and Parallel Systems*, December 1988.

[47] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 214–225, Boulder, CO, May 1990.

[48] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.